DEPARTMENT OF MATHEMATICS AND COMPUTING

V-M.Tech. (M&C)

Monsoon Semester 2022-2023

GPU Computing Lab

MCC302

LAB-3

Matrix-Matrix Multiplication

NAME: **ROMEO SARKAR**
ADMISSION NO.: **20JE0814**
DATE: **24-08-2022**

Experiment 1.1: Matrix-Matrix sum on GPU.

Objectives: Sum two matrices.

CUDA Sample Program:

```
#include <cuda_runtime.h>
#include <stdio.h>

void initialData (float *ip, const int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        ip[i] = i;
    }
    return;
}

void displayMatrix (float *A, int nx, int ny)
{
    int idx;
    for (int i = 0; i < nx; i++)
    {
        for (int j = 0; j < ny; j++)
        {
            idx = i * ny + j;
            printf ("%6.2f ", A[idx]);
        }
        printf ("\n");
    }
}

__global__ void sumMatrixOnGPU (float *MatA, float *MatB, float *MatC, int nx, int
ny)
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < nx)
    {
        for (int iy = 0; iy < ny; iy++)
        {
            int idx = iy * nx + ix;
            MatC[idx] = MatA[idx] + MatB[idx];
        }
    }
}

int main ()
{
    int nx = 4;
    int ny = 5;

    int nxy = nx * ny;
    int nBytes = nxy * sizeof (float);
    // malloc host memory
    float *h_A, *h_B, *h_C;
    h_A = (float *) malloc (nBytes);
    h_B = (float *) malloc (nBytes);
    h_C = (float *) malloc (nBytes);
    //
    initialData (h_A, nxy);
    initialData (h_B, nxy);
```

```
    float *d_MatA, *d_MatB, *d_MatC;
    cudaMalloc (&d_MatA, nBytes);
    cudaMalloc (&d_MatB, nBytes);
    cudaMalloc (&d_MatC, nBytes);

    cudaMemcpy (d_MatA, h_A, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy (d_MatB, h_B, nBytes, cudaMemcpyHostToDevice);


    int dimx  = 32;
    dim3 block (dimx, 1);
    dim3 grid ((nx + block.x - 1) / block.x, 1);

    sumMatrixOnGPU <<<grid, block>>> (d_MatA, d_MatB, d_MatC, nx, ny);

    cudaDeviceSynchronize ();
    cudaMemcpy (h_C, d_MatC, nBytes, cudaMemcpyDeviceToHost);
    displayMatrix (h_C, nx, ny);

    cudaFree (d_MatA);
    cudaFree (d_MatB);
    cudaFree (d_MatC);

    free (h_A);
    free (h_B);
    free (h_C);

    cudaDeviceReset ();
    return (0);
}
```

Output:

```
  0.00   2.00   4.00   6.00   8.00
 10.00  12.00  14.00  16.00  18.00
 20.00  22.00  24.00  26.00  28.00
 30.00  32.00  34.00  36.00  38.00
```

Lab Exercise 1.1: Write a CUDA program to demonstrate the followings:

1) Allocate Device Memory.
2) Transfer Data (Matrices A and B) from host to device.
3) Sum two matrices using 2D grid.
4) Transfer Data (Matrix C) from device to host.
5) Print the result in matrix format.

CODE:

```
// Lab Exercise 1.1

#include <cuda_runtime.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS 4
#define COLUMNS 8


__global__ void sum (double a[ROWS][COLUMNS], double b[ROWS][COLUMNS], double
s[ROWS][COLUMNS]);
void fill_data (void *p);
void display_matrix (void *p);


int main ()
{
    srand (time (NULL));

    double (*host_arr_a)[COLUMNS], (*host_arr_b)[COLUMNS], (*host_arr_c)[COLUMNS];
    host_arr_a = (double (*)[COLUMNS]) (malloc (ROWS * COLUMNS * sizeof
(double)));
    host_arr_b = (double (*)[COLUMNS]) (malloc (ROWS * COLUMNS * sizeof
(double)));
    host_arr_c = (double (*)[COLUMNS]) (malloc (ROWS * COLUMNS * sizeof
(double)));

    fill_data (host_arr_a);
    fill_data (host_arr_b);


// 1) Allocate Device Memory:
//[
    double (*device_arr_a)[COLUMNS], (*device_arr_b)[COLUMNS],
(*device_arr_c)[COLUMNS];
    cudaMalloc (&device_arr_a, ROWS * COLUMNS * sizeof (double));
    cudaMalloc (&device_arr_b, ROWS * COLUMNS * sizeof (double));
    cudaMalloc (&device_arr_c, ROWS * COLUMNS * sizeof (double));
//]


// 2) Transfer Data (Matrices A and B) from host to device
//[
    cudaMemcpy (device_arr_a, host_arr_a, ROWS * COLUMNS * sizeof (double),
cudaMemcpyHostToDevice);
    cudaMemcpy (device_arr_b, host_arr_b, ROWS * COLUMNS * sizeof (double),
cudaMemcpyHostToDevice);
//]


// 3) Sum two matrices using 2D grid
//[
    dim3 grid (ROWS, COLUMNS, 1);
    dim3 block (1, 1, 1);
    sum <<<grid, block>>> (device_arr_a, device_arr_b, device_arr_c);
    cudaDeviceSynchronize ();
//]


// 4) Transfer Result (Matrix C) from device to host
//[
    cudaMemcpy (host_arr_c, device_arr_c, ROWS * COLUMNS * sizeof (double),
cudaMemcpyDeviceToHost);
```

```
//]


// 5) Print the result in matrix format
//[
    std::cout << "matrix_a: " << std::endl;
    display_matrix (host_arr_a);
    std::cout << "matrix_b: " << std::endl;
    display_matrix (host_arr_b);
    std::cout << "matrix_c: " << std::endl;
    display_matrix (host_arr_c);
//]


    cudaFree (device_arr_a);
    cudaFree (device_arr_b);
    cudaFree (device_arr_c);

    free (host_arr_a);
    free (host_arr_b);
    free (host_arr_c);

    cudaDeviceReset ();

    return 0;
}

__global__ void sum (double a[ROWS][COLUMNS], double b[ROWS][COLUMNS], double
s[ROWS][COLUMNS])
{
    printf ("blockIdx=(%d,%d,%d)\n", blockIdx.x, blockIdx.y, blockIdx.z);
    if (blockIdx.x < ROWS)
    {
        if (blockIdx.y < COLUMNS)
        {
            s[blockIdx.x][blockIdx.y] = a[blockIdx.x][blockIdx.y] +
b[blockIdx.x][blockIdx.y];
        }
    }
    return;
}

void fill_data (void *p)
{
    // srand (time (NULL) + clock ());
    double (*mat)[COLUMNS] = (double (*)[COLUMNS]) (p);
    for (size_t i = 0; i < ROWS; i++)
    {
        for (size_t j = 0; j < COLUMNS; j++)
        {
            mat[i][j] = (double) (rand () % 100 - rand () % 100);
        }
    }
    return;
}

void display_matrix (void *p)
{
    double (*mat)[COLUMNS] = (double (*)[COLUMNS]) p;
    for (size_t i = 0; i < ROWS; i++)
    {
        for (size_t j = 0; j < COLUMNS; j++)
        {
            printf ("%7.2f ", mat[i][j]);
        }
        printf ("\n");
    }
}
```

Output:

```
blockIdx=(3,2,0)
blockIdx=(2,2,0)
blockIdx=(2,6,0)
blockIdx=(1,0,0)
blockIdx=(2,3,0)
blockIdx=(1,6,0)
blockIdx=(0,2,0)
blockIdx=(3,7,0)
blockIdx=(1,7,0)
blockIdx=(0,4,0)
blockIdx=(3,5,0)
blockIdx=(3,4,0)
blockIdx=(1,4,0)
blockIdx=(1,3,0)
blockIdx=(0,5,0)
blockIdx=(3,1,0)
blockIdx=(0,7,0)
blockIdx=(2,5,0)
blockIdx=(2,0,0)
blockIdx=(0,1,0)
blockIdx=(1,1,0)
blockIdx=(2,1,0)
blockIdx=(1,5,0)
blockIdx=(1,2,0)
blockIdx=(0,3,0)
blockIdx=(3,6,0)
blockIdx=(0,6,0)
blockIdx=(0,0,0)
blockIdx=(2,7,0)
blockIdx=(3,3,0)
blockIdx=(2,4,0)
blockIdx=(3,0,0)
matrix_a:
   2.00  -39.00  -74.00  -30.00   -2.00   22.00  -52.00    0.00
  30.00   32.00   -6.00   63.00   16.00  -58.00   60.00   10.00
 -32.00   10.00   28.00   18.00   98.00  -70.00  -19.00  -53.00
 -50.00  -39.00   35.00   -5.00   42.00  -39.00  -27.00  -39.00
matrix_b:
  58.00  -45.00   22.00   24.00   -6.00   56.00   26.00    0.00
  11.00    3.00  -67.00    2.00   21.00   54.00  -38.00    0.00
  72.00  -43.00  -16.00   59.00  -33.00   37.00  -73.00   20.00
 -25.00   68.00   -3.00   -1.00  -68.00   38.00    3.00   -8.00
matrix_c:
  60.00  -84.00  -52.00   -6.00   -8.00   78.00  -26.00    0.00
  41.00   35.00  -73.00   65.00   37.00   -4.00   22.00   10.00
  40.00  -33.00   12.00   77.00   65.00  -33.00  -92.00  -33.00
 -75.00   29.00   32.00   -6.00  -26.00   -1.00  -24.00  -47.00
```

Lab Exercise 1.2: Write a CUDA program to demonstrate:

1. Allocate Device Memory.
2. Transfer Data (Matrices A and B) from host to device.
3. Sum two matrices using 2D grid with different block sizes.
4. Transfer result (Matrix C) from device to host.
5. Print the result in matrix format.
6. Show the effect of block size and grid size in terms of total run time.

CODE:

```
// Lab Exercise 1.2

#include <cuda_runtime.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#define ROWS 64
#define COLUMNS 64

__global__ void sum (double a[ROWS][COLUMNS], double b[ROWS][COLUMNS], double
s[ROWS][COLUMNS]);
void fill_data (void *p);
void display_matrix (void *p);


int main ()
{
    srand (time (NULL));
    // clock_t c = clock ();
    double (*host_arr_a)[COLUMNS], (*host_arr_b)[COLUMNS], (*host_arr_c)[COLUMNS];
    host_arr_a = (double (*)[COLUMNS]) (malloc (ROWS * COLUMNS * sizeof
(double)));
    host_arr_b = (double (*)[COLUMNS]) (malloc (ROWS * COLUMNS * sizeof
(double)));
    host_arr_c = (double (*)[COLUMNS]) (malloc (ROWS * COLUMNS * sizeof
(double)));

    fill_data (host_arr_a);
    fill_data (host_arr_b);


// 1) Allocate Device Memory:
//[
    double (*device_arr_a)[COLUMNS], (*device_arr_b)[COLUMNS],
(*device_arr_c)[COLUMNS];
    cudaMalloc (&device_arr_a, ROWS * COLUMNS * sizeof (double));
    cudaMalloc (&device_arr_b, ROWS * COLUMNS * sizeof (double));
    cudaMalloc (&device_arr_c, ROWS * COLUMNS * sizeof (double));
//]


// 2) Transfer Data (Matrices A and B) from host to device
//[
    cudaMemcpy (device_arr_a, host_arr_a, ROWS * COLUMNS * sizeof (double),
cudaMemcpyHostToDevice);
    cudaMemcpy (device_arr_b, host_arr_b, ROWS * COLUMNS * sizeof (double),
cudaMemcpyHostToDevice);
//]


// 3) Sum two matrices using 2D grid with different block sizes
//[
    printf
("   \033[4mgridDim:\033[m        \033[4mblockDim:\033[m      \033[4mtime(s):\033
[m\n");

    for (int i = 1; i <= 1024; i *= 2)
    {
        int block_x = i, block_y = 1024 / i;
        dim3 block (block_x, block_y, 1);
        dim3 grid ((ROWS + block_x - 1) / block_x, (COLUMNS + block_y - 1) /
block_y, 1);

    // 6) show the effect of different block sizes
```

```
    //[
        printf ("%04d,%04d,%04d    %04d,%04d,%04d ", grid.x, grid.y, grid.z,
block.x, block.y, block.z);
        clock_t c = clock ();
        sum <<<grid, block>>> (device_arr_a, device_arr_b, device_arr_c);
        cudaDeviceSynchronize ();
        c = clock () - c;
        printf ("   %5.3f\n", ((float) (c)) / CLOCKS_PER_SEC);
        // }
    }
//]


// 4) Transfer Result (Matrix C) from device to host
//[
    cudaMemcpy (host_arr_c, device_arr_c, ROWS * COLUMNS * sizeof (double),
cudaMemcpyDeviceToHost);
//]


// 5) Print the result in matrix format
//[
    // std::cout << "matrix_a: " << std::endl;
    // display_matrix (host_arr_a);
    // std::cout << "matrix_b: " << std::endl;
    // display_matrix (host_arr_b);
    // std::cout << "matrix_c: " << std::endl;
    // display_matrix (host_arr_c);
//]


    cudaFree (device_arr_a);
    cudaFree (device_arr_b);
    cudaFree (device_arr_c);

    free (host_arr_a);
    free (host_arr_b);
    free (host_arr_c);

    cudaDeviceReset ();

    return 0;
}

__global__ void sum (double a[ROWS][COLUMNS], double b[ROWS][COLUMNS], double
s[ROWS][COLUMNS])
{
    int global_threadIdx = blockIdx.x * blockDim.x + threadIdx.x, global_threadIdy
= blockIdx.y * blockDim.y + threadIdx.y;
    // printf ("blockIdx=(%d,%d,%d);threadIdx=(%d,%d,%d)->{%d,%d,%d}\n",
blockIdx.x, blockIdx.y, blockIdx.z, threadIdx.x, threadIdx.y, threadIdx.z,
global_threadIdx, global_threadIdy, 0);
    if (global_threadIdx < ROWS)
    {
        if (global_threadIdy < COLUMNS)
        {
            for (int i = 0; i < 1024 * 1024 * 2; i++)
            s[global_threadIdx][global_threadIdy] =
a[global_threadIdx][global_threadIdy] + b[global_threadIdx][global_threadIdy];
        }
    }
    return;
}

void fill_data (void *p)
{
    double (*mat)[COLUMNS] = (double (*)[COLUMNS]) (p);
    for (size_t i = 0; i < ROWS; i++)
    {
        for (size_t j = 0; j < COLUMNS; j++)
```

```
        {
            mat[i][j] = (double) (rand () % 100 - rand () % 100);
        }
    }
    return;
}

void display_matrix (void *p)
{
    double (*mat)[COLUMNS] = (double (*)[COLUMNS]) p;
    for (size_t i = 0; i < ROWS; i++)
    {
        for (size_t j = 0; j < COLUMNS; j++)
        {
            printf ("%7.2f ", mat[i][j]);
        }
        printf ("\n");
    }
}
```

Outputs:

```
   gridDim:         blockDim:      time(s):
0064,0001,0001    0001,1024,0001    0.182
0032,0001,0001    0002,0512,0001    0.203
0016,0001,0001    0004,0256,0001    0.239
0008,0001,0001    0008,0128,0001    0.921
0004,0001,0001    0016,0064,0001    3.560
0002,0002,0001    0032,0032,0001    3.806
0001,0004,0001    0064,0016,0001    3.823
0001,0008,0001    0128,0008,0001    1.979
0001,0016,0001    0256,0004,0001    1.102
0001,0032,0001    0512,0002,0001    1.114
0001,0064,0001    1024,0001,0001    0.845
```

```
   gridDim:         blockDim:      time(s):
0064,0001,0001    0001,1024,0001    0.184
0032,0001,0001    0002,0512,0001    0.208
0016,0001,0001    0004,0256,0001    0.238
0008,0001,0001    0008,0128,0001    0.886
0004,0001,0001    0016,0064,0001    3.563
0002,0002,0001    0032,0032,0001    3.813
0001,0004,0001    0064,0016,0001    3.837
0001,0008,0001    0128,0008,0001    1.989
0001,0016,0001    0256,0004,0001    1.100
0001,0032,0001    0512,0002,0001    1.116
0001,0064,0001    1024,0001,0001    0.855
```